CS51 Final Project Writeup

Andrew Palacci

May 2022

Overview

As part of my CS51 final project, I was asked to extend the MiniML programming language to include additional features of my choosing. Initially, I chose to implement a lexical model for evaluating expressions in MiniML. I then moved on to add the float and string data types, along with several accompanying binary operations. Finally, I allowed for recognition of syntactical sugaring that, much like OCaml, allows functions to be curried when formed by Fun, Let, and Letrec expressions. This writeup will cover the functionality and implementations of these key extensions, alongside any important decisions and challenges I faced along the way.

Lexical Environment Scoping

In dynamically scoped environments, a function's definition depends on the environment that is being used when the function is called. However, in lexically scoped environments, the function's definition is kept constant - this is done by creating a closure, which saves a "snapshot" of the environment when the function is initially defined, for use whenever the function is called. In order to allow for this functionality, I had to augment three parts of my general evaluator to suit the lexical specification: Fun, App, and Letrec. In Fun, I returned a closure instead of just the function alone, and in App, I pattern-matched this closure and made sure to evaluate the function in the closure's environment. I found Letrec to be the trickiest - I carefully followed the instructions in the writeup, paying close attention to the possibility of an unbound variable and to the use of refs when mutating a value before completing the final evaluation. Here are my (abbreviated) implementations for these three expression types:

```
(* Fun: *)
| Fun _ -> if md = Lex then close exp env else Val exp
(* App: *)
| Closure (Fun (v, e), env_old) ->
                    let vq = ev_inner e2
                    in let ext = extend env_old v (ref vq)
                    in eval_aux Lex e ext
| _ -> raise (EvalError "invalid input (not a closure/function)")))
(* Letrec: *)
| Lex -> let x = ref (Val Unassigned)
            in let env_new = extend env v x
            in let vd = eval_aux Lex e1 env_new
            in (match vd with
                | Val Var _ -> raise (EvalError "letrec unbound variable")
                | _ -> x := vd; eval_aux Lex e2 env_new))
```

Finally, I made sure to test thoroughly, using tests like the following, which checked that lexical and dynamic output were not always equivalent:

```
(* let x = 1 in let f = fun y -> x + y in let x = 2 in f 3 ;; *)
(* should evaluate to 4 with eval_s/eval_l and 5 with eval_d *)
let comp1 = Let("x", Num(1), Let("f", Fun("y", Binop(Plus, Var("x"), Var("y"))),
        Let("x", Num(2), App(Var("f"), Num(3)))))

(* let x = 5. in let f = fun y -> x in let x = 3. in f 6 ;; *)
(* slightly confusing, but should evaluate to 5. in lex/sub and 3. in dyn *)
let comp2 = Let("x", Float(5.), Let("f", Fun("y", Var("x")),
            Let("x", Float(3.), App(Var("f"), Num(6)))))

(* tests letrec and application using compound tests *)
let eval_compound_test (ev : (expr -> env -> value)) (md : model) : unit =
    unit_test (ev comp1 (empty ()) =
            if md = Dyn then Val (Num 5) else Val (Num 4))
            "eval comp1 empty env";
    unit_test (ev comp2 (empty ()) =
            if md = Dyn then Val (Float 3.) else Val (Float 5.))
            "eval comp2 empty env";
```

(Note that for testing, I declared expressions for testing separately, and also tested each evaluator by taking an evaluator and model as input)

Floats

After lexical scoping, my next extension involved introducing floats to MiniML. This required gaining some understanding in parsing via the two files miniml_lex.mll (lex) and miniml_parse.mly (parse). I realized that I needed to use lex to create a regular expression (regex) for recognizing floats, and then tokenize them to be passed into the parse, where they were dealt with identically to INTs. My work in parsing can thus be summarized with the following:

```
let exp = ['e' 'E'] ['-' '+']? digit*
let frac = '.' digit*
let fl = digit+ frac? exp?

rule token = parse
    | fl as ifloat
        { let num = float_of_string ifloat in
          FLOAT num
        }
```

After parsing floats and creating a complementary Float expression (and to_string functionality) in expr.ml, I lastly had to make sure evaluations using floats would work. This is where I reached one major decision in my extensions - I wasn't sure whether to make MiniML strongly typed, like OCaml, or allow for inter-type evaluations (such as comparing or adding floats with ints). As someone who is partial to more flexibility when programming, I decided to make MiniML weakly typed by allowing most binops to operate on both floats and ints (namely Plus, Minus, Times, Divide, Equals, LessThan, and GreaterThan). Although this resulted in a plethora of legal binops, it saved me from implementing float-specific operations and also provided an interesting foray away from OCaml. Once again, I made sure to thoroughly test floats by considering and testing each possible float binop in eval_binop_test ().

Strings

Similarly to Floats, I recognized that the creation of a String data type would require additional modification of the lex and parse files. I again began by creating a regex for strings:

```
let string = ['"'] [^ '"']* ['"']
```

Then I recognized that this regex would store the quotes as part of the string during parsing, so I removed the quotes by taking a substring for the entire string except the first and last index, and passed this in as a token to be parsed:

```
| string as str
    { STRING (String.sub str 1 (String.length str - 2))
    }
```

Finally, I parsed strings similarly to floats in the parse file and added all necessary expressions and to_string functionality in expr.ml. In terms of operating on strings, this will be covered in the subsequent section titled "Binops".

Binops

In order to account for my two new data types (Floats and Strings) and simply add more functionality to MiniML, I decided to introduce a few new binops, and also extend the uses for current binops. More specifically, I added GreaterThan, Divide, and the two logical conjunctions And and Or. This was done by introducing the symbols ">", "/", "&&", and "||" as tokens in lex, and then parsing them as new Binops in parse. Then, in my eval_binop helper function in evaluation.ml, I added the ability to compare floats to floats or ints using LessThan, GreaterThan, and Equal, and I also allowed Floats to be operated on using Plus, Minus, Times, Divide with other Floats and also with Nums. All of the float-int interactions were facilitated by casting ints to floats using float_of_int, with an example below:

```
| GreaterThan, Val Num v1, Val Float v2 -> Val (Bool ((float_of_int v1) > v2))
```

The boolean "and" and "or" conjunctions were implemented using the same operators in OCaml:

```
| Or, Val Bool v1, Val Bool v2 -> Val (Bool (v1 || v2))
```

Finally, MiniML also allows for String comparison and concatenation, with concatenation using the "+" operator:

```
| GreaterThan, Val String v1, Val String v2 -> Val (Bool (compare v1 v2 > 0))
| Plus, Val String v1, Val String v2 -> Val (String (v1 ^ v2))
```

This concatenation method is intended to emulate Python, one of my favorite languages to code in, which also uses the "+" operator to concatenate strings. As before, all of these new binops were thoroughly tested in eval_binop_test () using 37 expressions and corresponding unit tests.

Syntactic Sugar

The last extension I made to MiniML is recognizing syntactic sugar that represents curried functions. One such example is the following function declaration:

```
let f x y = x + y in f 3 4 ;;
```

This can alternatively be written as:

```
let f = fun x -> fun y -> x + y in f 3 4 ;;
```

Syntactic sugaring is what allows these two expressions to be processed in the same way. Fortunately, because expr is an algebraic data type in MiniML, it is straightforward to have functions nested within each other, meaning that currying is built-in, and meaning that only parsing is necessary to permit this syntactic sugar. The first step was introducing a new nonterminal in parse as follows:

```
idlist:  ID idlist              { $1 :: $2 }
       | ID                     { [$1] }
```

Note that this parallels the formatting of the exp nonterminal in parse, with the exception that it is parsed as a list instead of an App expression. Realizing I could parse idlist into an OCaml list construct was one of my greatest challenges, as I had been attempting to put it in expression format, as with the exp nonterminal. However, this was crucial because it then allowed me to create a function in OCaml to iterate over a list of ids (namely, an idlist) and convert it to a nested Fun expression as such:

```
let rec lst_to_fun (lst : string list) (exp : expr) : expr =
    match lst with
    | [] -> exp
    | hd :: tl -> Fun(hd, lst_to_fun tl exp) ;;
```

Finally, I augmented expnoapp to include the following three cases, allowing for the syntactic sugar of curried functions:

```
| LET ID idlist EQUALS exp IN exp   { Let($2, lst_to_fun $3 $5, $7) }
| LET REC ID idlist EQUALS exp IN exp { Letrec($3, lst_to_fun $4 $6, $8) }
| FUNCTION ID idlist DOT exp   { Fun($2, lst_to_fun $3 $5) }
```

Here is an example of the trivial evaluator running in MiniML, to show how this parsing works to create a curried function:

```
<== let f x y = x + y in f 3 4 ;;
==> Let("f", Fun("x", Fun("y", Binop(Plus, Var("x"),
    Var("y")))), App(App(Var("f"), Num(3)), Num(4)))
```

As usual, I tested this extension by adding numerous tests on externally-declared expressions that were created using syntactic sugar. See below for another example:

```
(* let ord a b c = (a < b) && (b < c) in ord 1 2 3 ;; *)
(* should evaluate to Val Bool True (ord checks if a < b < c) *)
let sugar2 = Let("ord", Fun("a", Fun("b", Fun("c", Binop(And, Binop(LessThan,
            Var("a"), Var("b")), Binop(LessThan, Var("b"), Var("c")))))),
            App(App(App(Var("ord"), Num(1)), Num(2)), Num(3)))

unit_test (try ev sugar2 (empty ()) =
            Val (Bool true) with (EvalError "val not found") -> md = Dyn)
            "eval sugar2 empty env";
```

Note that this unit test also returns true if the "val not found" error is raised in the dynamic environment, since syntactic sugaring does not match up with the rules for a dynamic environment.

Additional Notes

One final (albeit small) addition I made is to print closures in miniml.ml, using the env_to_string function made in an earlier stage of the project:

```
| Closure (resexp, env) ->
       printf "==> %s, {%s}\n"
       (Ex.exp_to_concrete_string resexp)
        (Env.env_to_string env)
```

I also added an expressions.ml file to store all of the expressions I used during testing, and created numerous testing functions in tests.ml that I called later on (see the Lexical section for one such example). Thank you for reading my writeup, I hope you enjoyed it!